IRQ-Tape Loader by Laze Ristoski (Cybertronic Studios)

(This rant is mirrored from Cadaver's site)

Welcome to my first tutorial written for Covert BitOps about IRQ-Tape Loaders.

0. My history

- 1986 This year I had my first C64 which was soon sold. Reason: I was 1 year old then. :)
- 1991 Got another C64C. I had no games, so I had to type them from various books.
- 1992 I started learning BASIC. This year I founded my own group 'Ninja-Soft Technologies'
- 1993-1996 BASIC, BASIC, BASIC...
- 1997 Started learning assembler. Never touched BASIC since then.
- 1998-2001 Coding games, utilities, intros, note writers, crunchers, IRQ loaders, etc.
- 2002 Changed the name of the group into 'Cybertronic Studios'

There was PC stuff mean while, but that is out of scope.

1. What is an IRQ-Tape Loader?

These loaders are very common in tape originals. They show a bitmap and play music while the game is loading.

Advantages of IRQ based loaders:

- You can have music playing while loading. This is the only way to achieve it. Although, you don't need an IRQ if you just want to show a bitmap.

Disadventages of IRQ based loaders:

- Slower loading than normal TURBO TAPE files.
- A little bit more difficult to write than normal (non-IRQ) TURBO TAPE loader.

2. The I/O ports

First we should understand how to communicate with the tape recorder. Unlike normal tape recorders, this one uses a digital record which is consisted of pulses. To generate a pulse, all we need to do is to invert bit 3 of the DR register.

Here is a very simple routine to achieve this:

```
LDA $01
EOR #$08
STA $01
```

But, how do we read this? The read line is connected to CIA1's FLAG line. A special feature of this line is that it is sensitive to Negative Edge (changing from 1 to 0). Commodore's tape recorder has built-in invertor which inverts this line on every pulse. So we can't read every pulse, but every second pulse. We can read the FLAG line from $DC0D, but we don't have to since we are using an IRQ to achieve this.

3. The pulses

You maybe noticed that there is nothing we can use for handshaking. Yeah, it's true, the tape is an asynchronous device. But how do we write 0 or 1 to the tape? Actually, we measure the pause between each pulses. Shorter pause is 0 and longer pause is 1.

From now on, I'll use the term pulse to refer the pause between each succesive pulses.

Here are the pulses used in normal turbo tape:

```
0 - 216 cycles.
1 - 326 cycles.
```

When reading, the timer should be somewhere between these two values. The actual value for the timer is 263 cycles.

But, these values are inadequate since we are showing a picture and playing music. Also note the badlines. So we need more tolerant values.

Here are the values I use for an IRQ-Loader:

```
0 - 446 cycles.
1 - 668 cycles.
Timer - 576 cycles.
```

I mentioned that we can read each second pulse (I don't mean the pause), so when writing to tape we have to divide each pause on two parts. (e.g. 446 cycles = 2 x 223)

4. Asynchronous transfer

Since we have no handshaking lines to use, there has to be another way to stay synchronised. The method is the same as normal turbo tape. We write a series of bytes with value 2 (the kernal routines use another way, which is out of scope). After these bytes, we write a count-down (bytes with values from 9 to 0) used for checking the synchronisation. Then goes the actual data like start address, end address and the program itself. It's important to have both start and end address, so the loader knows how many bytes to load and then read the checksum. In the example below, instead of a count-down,

there is a Synchro Check Byte (value = $5A).

5. The Checksum

It would be fine if we have a way to detect possible errors in transmission. These errors can be caused by wrong azimuth. A way exists and it's very simple. Before loading we initialise a variable to 0 and we add the bytes here. Then we check this with the checksum from the tape. The checksum is always the last byte of the file. An important thing is to know that we can not use OR operation for this as once the value will become $FF it will remain. So we will use EOR (exclusive or) for adding the byte values.

6. The registers

To understand the example below, you need at least a basic knowledge of Assembler and the registers used for communication.

```
$01 (Data Register - DR)
bit 7,6 - undefined
bit 5 - Cassette Motor Control (0=on, 1=off)
bit 4 - Cassette Switch Sense (1=switch closed)
bit 3 - Cassette Data Output Line (invert to write a pulse)
bit 2 - CHAREN (0=char rom in) [Not Needed]
bit 1 - HIRAM (0=kernal rom out) [Not Needed]
bit 0 - LORAM (0=basic rom out) [Not Needed]
```

```
$DC0D (CIA1 Inrerrupt Control Register - ICR)
bit 7 - interrupt occured (1 if any interrupts)
bit 6,5 - unused
bit 4 - FLAG
bit 3 - SP [Not Needed]
bit 2 - Alarm [Not Needed]
bit 1 - Timer B
bit 0 - Timer A
```

7. An example

The best way to understand this stuff is an actual example. Here are the routines used for save and load.

```
- SAVE

    SLBS     = $AC
    SHBS     = $AD
```

```
    SLBE     = $AE
    SHBE     = $AF
    LLBS     = $02
    LHBS     = $03
    LLBE     = $04
    LHBE     = $05
    SYNCLEN  = $AB
    CHECKSUM = $D7
    BUFFER   = $BD
    BITCNT   = $A3

    SEI
    JSR $F838    ; Show 'Press record and play on tape' message and wait
    LDA #0          ; until the user does that
    STA $90        ; Initialise system's status variable (0 - no error)
    LDA # [SLBS]    ; LO-Byte of the save add. (This is where the prg. we
    STA SLBS        ; are saving is in memory)
    LDA # [SHBS]    ; HI-Byte...
    STA SHBS
    LDA # [SLBE]    ; LO-Byte of end address
    STA SLBE
    LDA # [SHBE]    ; HI-Byte...
    STA SHBE
    LDA # [LLBS]    ; LO-Byte of the load add. (Where the prg. should be
    STA LLBS        ; loaded)
    LDA # [LHBS]    ; HI-Byte...
    STA LHBS
    LDA # [LLBE]    ; LO-Byte of end address
    STA LLBE
    LDA # [LHBE]    ; HI-Byte...
    STA LHBE
    LDA $D011    ; Disable VIC-II (This has to be done because of
    AND #$EF        ; badlines
    STA $D011
    LDA $01        ; Disable BASIC ROM and enable cass. motor
    AND #$DE
    STA $01
    LDX #$00    ; Make a short pause so the cass. motor can reach the
    LDY #$00        ; needed speed
PAUSE   DEY
    BNE PAUSE
    DEX
    BNE PAUSE
    LDA #$02    ; Length of synchronisation record
    STA SYNCLEN
    JSR SYNCHRO    ; Write synchronisation
    LDY #$00
    LDX #$1E    ; X is used for the Time Constant (TC) needed for the
CONT    LDA $LLBS,Y     ; pulse length
    JSR SAVE
    LDX #$22    ; TC
```

```
    INY
    CPY #$04
    BNE CONT
    LDY #$00
    STY CHECKSUM
    LDX #$22    ; TC
MORE    LDA (SLBS),Y    ; Get byte of the program
    JSR SAVE    ; Save the byte on tape
    LDX #$1F    ; TC
    INC SLBS
    BNE INCR
    INC SHBS
    DEX        ; TC = TC - 2 (because INC SHBS steals some cycles)
    DEX
INCR    LDA SLBS    ; All bytes saved ?
    CMP SLBE
    LDA SHBS
    SBC SHBE
    BCC MORE    ; No, save more
    LDX #$20    ; TC
    LDA CHECKSUM    ; Save the CHECKSUM
    JSR SAVE
    LDA $01        ; Disable cass. motor and enable BASIC ROM again
    ORA #$21
    STA $01
    LDA $D011    ; Enable VIC-II
    ORA #$10
    STA $D011
    LDA #$01    ; CAS1 (This is a system variable. 1 - cass. motor off)
    STA $C0
    CLI
    RTS

SYNCHRO LDY #$00
    LDX #$25    ; TC
NEXT    LDA #$02    ; 2 is the synchro byte
    JSR SAVE    ; Save it
    LDX #$23    ; TC
    INY
    BNE NEXT    ; Save more synchro bytes
    LDX #$22    ; TC
    DEC SYNCLEN
    BNE NEXT    ; Save more synchro bytes
    LDX #$22    ; TC
    LDA #$5A    ; Synchro check byte
    JSR SAVE    ; Save it
    RTS

SAVE    STA BUFFER    ; Store the byte in a buffer
    EOR CHECKSUM
    STA CHECKSUM    ; Calculate checksum
```

```
    LDA #$08
    STA BITCNT    ; Initialise bit counter
GO  ASL BUFFER
    JSR WAIT    ; Make the needed delay
    DEC BITCNT
    BNE GO
    RTS

WAIT    LDA #$06    ; Used for inverting the cass. data output line
    JSR WAIT2
    LDX #$2A    ; TC
    LDA #$0E    ; Used for inverting the cass. data output line
    JSR WAIT2
    LDX #$26    ; TC
    RTS

WAIT2   DEX
    BNE WAIT2    ; Make the needed delay
    BCC B0        ; If recording a 0 bit, no more delay needed
    LDX #$15    ; TC
B1  DEX
    BNE B1        ; Make more delay if bit 1
B0  STA $01        ; Record a pulse on tape
    RTS
```

Some explanations: SLBS, SHBS, SLBE, SHBE must be on zero page and take successive bytes in memory. The same applies for LLBS, LHBS, LLBE, LHBE. There are save and load addresses. You might wonder why. This is because when saving we can have the pic. for example at addr. $6000, but when loading we might need it at addr. (e.g. $E000). TC is the time constant. This is used to make the pause always the same, because different op-codes are processed which take different nr. of cycles.

```
- LOAD

    SEI
    LDA #$00    ; Set black screen
    STA $D020
    STA $D021
    LDA #$00    ; Set start and end addresses of the bitmap
    LDX #$E0
    STA $AC
    STX $AD
    LDA #$40
    LDX #$FF
    STA $AE
    STX $AF
    LDY #$00    ; Clear the bitmap
CLRMAP  LDA #$00
    STA ($AC),Y
    INC $AC
    BNE INCR1
```

```
    INC $AD
INCR1   LDA $AC
    CMP $AE
    LDA $AD
    SBC $AF
    BCC CLRMAP    ; If not all cleared, clear more
    LDA #$00
    STA $D011    ; Disable VIC-II (because the bitmap is still not loaded)
    LDA $01        ; Disable KERNAL and BASIC ROM and enable cass. motor
    AND #$DD
    STA $01
    LDA #$00    ; Disable VIC-II interupts
    STA $D01A
    LDA $D019    ; Acknowledge VIC-II interupts
    STA $D019
    LDA #$7F    ; Disable CIAs interupts
    STA $DC0D
    STA $DD0D
    LDA #$90    ; Enable FLAG (cass. input) interrupt
    STA $DC0D
    LDA #$40    ; Set timer value to 576 cycles
    LDX #$02
    STA $DC06
    STX $DC07
    LDA $DC0D    ; Acknowledge CIAs interrupts
    LDA $DD0D
    LDA #<NMI    ; Initialise NMI vector
    LDX #>NMI
    STA $FFFA
    STX $FFFB
    LDA #<IRQ    ; Initialise IRQ vector
    LDX #>IRQ
    STA $FFFE
    STX $FFFF
    LDA #$08    ; Initialise BIT COUNTER
    STA BITCNT
    LDA #$00    ; This flag is used as MUSIC PLAY ON/OFF (0-off) because
    STA $FE        ; music is still not loaded
    STA $FF        ; Flag for the part that is loading
    CLI
NEXT    JSR PRGLOAD    ; This routine waits until a part is loaded
    INC $FF        ; Increment loading part flag
    LDA $FF
    CMP #$01    ; The music has been loaded
    BNE P1
    LDA #$00    ; Initialise music
    TAX
    TAY
    JSR $7000    ; The init add. of the music I'm using
    LDA #$01    ; Set MUSIC PLAY flag. The music is loaded now, so it
    STA $FE         ; can be played
```

```
    JMP NEXT    ; Load next part
P1  CMP #$04    ; The BITMAP (screen, color and bitmap) has been loaded
    BNE P2
    LDA #$94    ; Change VIC-II bank ($C000-$FFFF)
    STA $DD00
    LDA #$08    ; Set bitmap memory
    STA $D018
    LDA #$D8    ; Set multi-color mode
    STA $D016
    LDA #$3B    ; Set bitmap display mode
    STA $D011
    JMP NEXT    ; Load next part
P2  CMP #$05    ; First part of game has been loaded (start - $7000)
    BNE P3
    LDA #$00    ; Disable music (the game will overlap it)
    STA $FE
    LDX #$18    ; Disable SID voices
    LDA #$00
CLSID   STA $D400,X
    DEX
    BPL CLSID
    JMP NEXT    ; Load next part
P3  CMP #$06    ; All parts loaded ?
    BNE NEXT    ; No, load next part.

    SEI
    LDA $01        ; Disable cass. motor and enable ROMs again
    ORA #$22
    STA $01
    LDA #$01    ; CAS1 (This is a system variable. 1 - cass. motor off)
    STA $C0
    LDA #$00    ; SID volume = 0
    STA $D418
    JSR $E5A8    ; Kernal routine: Initialise VIC-II
    LDA #$97    ; Change VIC-II bank ($0000-$3FFF)
    STA $DD00
    CLI
    JMP $0810    ; Jump to program's start address (change this if
                       ; another is used)
PRGLOAD LDA #$00    ; Flag: synchronisation reached (0-no, 1-yes)
    STA $FC
    STA $FD        ; Flag: part loaded (0-no, 1-yes)
    STA CHECKSUM    ; Initialise checksum
WAIT    LDA $D012    ; A routine for playing the music
    CMP #$7C
    BMI NOMUSIC
    CMP #$84
    BPL NOMUSIC
    LDA $FE        ; MUSIC PLAY Flag (1-play, 0-no)
    BEQ NOMUSIC
    JSR $7003    ; The play add. of the music I'm using
```

```
NOMUSIC LDA $FD     ; Is the part loaded ?
    BEQ WAIT
    RTS

IRQ PHA
    TXA
    PHA
    TYA
    PHA
    LDA $DC0D    ; See if counter has finished counting (no = bit 0,
    LDX #$19    ; yes = bit 1). Start counter again
    STX $DC0F
    LSR A        ; Get counter finished flag in CARRY
    LSR A
    ROL $BD        ; Put the bit in buffer
    LDA $BD
    INC $D020    ; Make the screen flash
    DEC $D020
    LDX $FC
    BEQ GETSYNCHRO    ; If no synchronisation, reach it
    DEC BITCNT
    LDX BITCNT
    BEQ CONT
    JMP RET
CONT    LDX #$08
    STX BITCNT
    LDX $FC
    CPX #$01
    BEQ TEST    ; Synchro Check Byte
    CPX #$02
    BEQ ADDRESS    ; Start and end address
    CPX #$03
    BEQ PROGRAM    ; Main part
    CPX #$04
    BEQ CHKSUM    ; Checksum

GETSYNCHRO
    CMP #$02    ; 2 is the synchro byte
    BNE RET
    INC $FC
    JMP RET

TEST    CMP #$5A    ; $5A is the synchro check byte
    BEQ GOOD
    LDA #$00    ; Synchronisation invalid. Retry
    STA $FC
    JMP RET
GOOD    INC $FC
    LDA #$AC    ; $00AC is the place in memory where we keep the start
    LDX #$00        ; and end address
    STA $02
```

```
    STX $03
    JMP RET

ADDRESS LDY #$00
    STA ($02),Y
    INC $02
    LDA $02
    CMP #$B0    ; All bytes of the address read ?
    BNE RET
    INC $FC
    JMP RET

PROGRAM LDY #$00
    STA ($AC),Y    ; Store byte in memory
    EOR CHECKSUM    ; Calculate checksum
    STA CHECKSUM
    INC $AC
    BNE INCR2
    INC $AD
INCR2   LDA $AC
    CMP $AE
    LDA $AD
    SBC $AF
    BCC RET
    INC $FC
    JMP RET

CHKSUM  CMP CHECKSUM
    BNE ERROR
    LDA #$00    ; Set no synchronisation (for the next part)
    STA $FC
    LDA #$01    ; Part loaded flag
    STA $FD
    JMP RET

ERROR   LDA $01        ; Disable cass. motor
    ORA #$20
    STA $01
    LDA #$0F    ; Set lt.grey border indicating load error
    STA $D020
    LDA #$00    ; Set SID volume to 0
    STA $D418
LOOP    JMP LOOP    ; Endless loop

RET LDA $DC0D    ; Acknowledge interrupt
    PLA
    TAY
    PLA
    TAX
    PLA
NMI RTI
```

Some explanations: When playing the music, the program checks to see if the raster reg. is between $7C and $84. We need to have more lines, because of the interrupt. If we check for only one specific raster line the music will play slow as there will be pulses that will exactly happen on this line. If the game exceeds $7000 (because the music starts here) it should be divided in two parts. First part: start address to $7000. Now the loader disables the music. Second part: $7000 to end address. The loader assumes the play button is pressed as the loader itself is saved on the begining of the record and has an autostart. You might need to add a code that will restore the original values of the vectors used for autostart. If the game reaches $C000, you should turn the screen off, when it reaches $7000 (together with the music). This is because the BITMAP is located at $C000.

- The routines above are given just for educational purposes. If you want to use the loader, download the source in JC-ASS together with the instructions My MULTI-LOAD can be freely used for commercial purposes, but don't forget to give me a credit. :)
- I was told that some music routines play faster than normal when used in the loader. I've never noticed such a problem. However, if this happens just reduce the number of raster-lines in the loader routine (eg. $7C to $80, instead of $7C to $84).
- Finaly, huge thanks goes to Joe Dixon and Richard Bayliss for testing the routines, and reporting some bugs I made when I retyped them. Anyway, the JC version has always been bug-free :)

Laze Ristoski

Additional note:

The JC-ASS and MULTI-LOAD source no longer exists (unless anyone still has it). So I decided to make a slightly modified tape master using the example source. Binaries and C64Studio/ACME source code, with instructions are available from Laze's Turbo Saver+Loader/Tape Master Source + Binary files

From:
https://codebase64.org/ - **Codebase 64 wiki**

Permanent link:
https://codebase64.org/doku.php?id=base:irq-tape_loader&rev=1523010019

Last update: **2018-04-06 12:20**